Software-Defined Radio

Lab 1: Getting in Touch with GNU Radio and the USRP

Version 0.5

Anton Blad, Mikael Olofsson, and Emil Björnson

November 16, 2015

1 Introduction

The purpose of this lab exercise is to give a first hands-on impression of signal processing and communication using GNU Radio and the USRP. It covers the basics of the GNU Radio flowgraph descriptions and gives an understanding of the transmission and reception paths of the USRP.

You will start with a very simple flowgraph that produces a sine wave, alters the amplitude, and plays the signal on the speakers. You will then alter the flowgraph to instead send the signal to the USRP, either in the baseband or at a carrier frequency, and visualize the output using an oscilloscope. When you have a basic grasp of the processing that happens in the transmission chain, you will write a new flowgraph that receives a signal using the USRP and shows its discrete Fourier transform (based on FFT) on the screen. You will also play the received signal on the speakers. Finally, you will replace the sine wave source with a wave file reader, and add configurable filters at the receiver side with which it is possible to change the bass and treble levels. Along the way, you will also see how you can present graphs of signals using a GUI.

2 Preparations

To finish the lab exercise on time, the following preparation tasks should be done in advance.

1. Read through the entire document. Pay particular attention to the flowgraph concept and read up on how to connect blocks using Python commands.
2. Read up on what interpolation and decimation mean in the context of signal resampling.
3. The signal source generates samples at a rate of 48 kHz, while the D/A conversion rate at the USRP is fixed at 128 MHz. Choose the interpolation rate `usrp_interp` of the USRP transmitter, according to the guidelines in Section 4.1. Based on that, compute the interpolation rate `fir_interp` and the decimation rate `fir_decim` of the rational resampler on the host computer. Notice that all involved sampling frequencies must be integers and all interpolation and decimation rates have to be integers.
4. The USRP receiver has an A/D conversion rate of 64 MHz. Choose a suitable USRP decimation factor, according to the guidelines in Section 4.4. Compute the additional interpolation and decimation that is needed at the host to obtain a signal with the sampling rate 48 kHz.

3 Setup

The following hardware is needed for this lab exercise:

- A computer with USB 2.0.
- Speakers connected to the line-out.
- A USRP with an LFTX/LFRX daughterboard pair at the A side.
- An oscilloscope with four channels (two could do the trick).

You need GNU Radio installed. The lab exercise is tested and ensured to work with GNU Radio 3.4.2, and assumes that it is installed in `/usr/local/gnuradio-3.4.2`.

3.1 Initialization

If you are doing this lab exercise in CommSys' research lab:
Log in to one of the lab computers as the user *SDR lab user*. Your lab teacher will give you the password. Your lab computer is prepared with all needed files in the directory

```
/home/sdrlabuser/labfolder/lab1
```

Open a terminal and move into that directory using `cd`. You do not need to worry about making changes to files there, because all changes will be reverted for the next lab occasion.

If you are not following these instructions in CommSys' research lab:
Open a terminal. Create a directory `labfolder` somewhere and cd into it. Download and extract the accompanying lab files by executing the following commands:

```
$ wget http://www.commsys.isy.liu.se/SDR/labs/gr/lab1.tar.gz
$ tar zxf lab1.tar.gz
$ cd lab1
$ wget http://www.commsys.isy.liu.se/SDR/labs/gr/test.wav
```

All files needed in this lab are now in the directory `lab1` that you just created.

3.2 First Tests

Open the file `base.py`, which contains the Python code for the flowgraph shown in Figure 1. This simple flowgraph is composed of the following three blocks:

- A signal source `gr.sig_source_f`, producing a stream of floats (e.g., representing a sine wave);
- A multiplier `gr.multiply_const_ff`, accepting float inputs and producing float outputs;
- A signal sink `audio.sink`, accepting a stream of floats and sending them to the speakers.

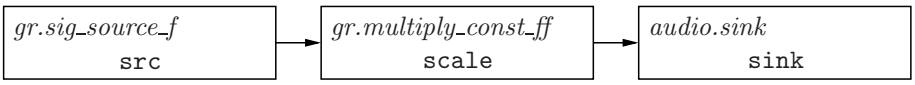

Figure 1: Simple flowgraph generating a sine signal and playing it on the computer speakers.

Study the code and then try to run it, which should play a tone on the computer's speakers. You run it from the terminal by writing

```
$ ./base.py
```

You stop it using Ctrl-c.

Note: Even though the `gr.sig_source_f` block is created with a sampling frequency of 48 kHz, this does not ensure that samples are produced at this rate. The block has no notion of time, and the `sampling_freq` argument is only used to determine the successive phase shifts between the samples. The `audio.sink` block is the only block affecting the processing rate in the system, by having a fixed rate at which samples can be fed to the sound card.

Study the Python code and make sure that you understand how the blocks in Figure 1 are created and connected. Answer the following:

- What is the frequency of the sine wave? How do you change it?
- What is the amplitude of the sine wave? How do you change it?

There is also the file `base_gui.py`, which does the same thing as `base.py`, but also has a GUI with two graphs. The upper graph shows the signal in the time domain the same way as an oscilloscope would do. The lower graph shows a power spectrum of the signal in dB scale based on an FFT. Open that file to see how that is accomplished. The differences compared to `base.py` are clearly indicated in the code using comments.

4 First Experiments on the USRP

The first step is to connect the USRP. Make sure that an LFTX/LFRX daughterboard pair is connected to the A side of a USRP unit. Then connect a power cable to the USRP and connect the USRP to the PC using a USB cable. You should see a LED blinking with approximately 3 Hz frequency on the USRP motherboard, right next to the power connector. This indicates that the USRP is ready to be initialized. Ensure that the computer can find the USRP by running

```
$ /usr/local/gnuradio-3.4.2/bin/lsusrp
```

The command should print the USRP serial number and the connected daughterboards. There might also be a list of other non-connected devices. After this, the LED blinks with approximately 1 Hz frequency. This indicates that the USRP has been initialized.

4.1 Interfacing to the USRP

In the example files `base.py` and `base_gui.py`, the signal source is generating samples at a rate of 48 kHz, which is well-suited for the speakers but much lower than the sample rate that the USRP supports. The D/A conversion rate at the USRP is fixed at 128 MHz. The USRP supports any interpolation factor $f = 4n$, where $n \in \{1, \ldots, 128\}$. This is not enough to interpolate a signal from 48 kHz to 128 MHz, thus we need to choose a suitable interpolation factor for the USRP

and in addition also upsample the signal in the host computer before sending it to the USRP. In our case, the interpolation rate needs to be rational since $48 \cdot 10^3$ does not divide $128 \cdot 10^6/4$. We will therefore use the block `blks2.rational_resampler`, which combines interpolation and decimation to perform rational resampling.

4.2 Building a Transmitter

Begin by saving the file `base.py` with a new name `usrp_tx.py`, and do the changes in this file. Alternatively, if you want a GUI, do this based on the file `base_gui.py`. Make sure that the new file has permission to be executed (you can change the permissions for a file using `chmod` in the terminal if needed). Then perform the following changes to the code.

To interface to the USRP in the Python code and to be able to use `blks2.rational_resampler`, first import the relevant modules:

```
from gnuradio import usrp, blks2
```

Next, remove the `audio.sink` instance and the `self.connect` command, and replace it by the following code. The first three lines contain values that you should compute as a preparation to this lab exercise. These parameters should all have integer values. As a result of the calculations, some of them may be represented as floats, though, and need to be transformed into integers, which is done using the Python internal function `int`.

```
usrp_interp = <choose value>
fir_interp = <compute/choose value>
fir_decim = <compute/choose value>

carrier = 0

self.fir = blks2.rational_resampler_fff(fir_interp, fir_decim)
self.uscale = gr.multiply_const_ff(2**15)
self.f2c = gr.float_to_complex()
self.u = usrp.sink_c(interp_rate = usrp_interp, nchan = 1)

subdev_spec = (0,0)
subdev = self.u.selected_subdev(subdev_spec)
self.u.set_mux(self.u.determine_tx_mux_value(subdev_spec))
self.u.tune(0, subdev, carrier)
subdev.set_gain(0)

self.connect(self.src, self.scale, self.fir, self.uscale,
             self.f2c, self.u)
```

In this code, the line `blks2.rational_resampler_fff(fir_interp,fir_decim)` produces a rational resampler block that also lowpass filters the resulting signal with a normalized cut-off frequency of 0.4 relative to the resulting sampling frequency. The parameters are the interpolation and decimation rates of the rational resampling. The rational resampler is followed by a scaling by 2^{15} and conversion to complex, which is necessary because the USRP accepts complex samples in the range $[-2^{15}, 2^{15} - 1]$. When the USRP is instantiated, the interpolation rate and number of transmitter paths are set. The next five lines obtain a reference to the daughterboard connection

at side A, configures the FPGA multiplexer (mux) to use its DAC path for the incoming samples, tunes the carrier frequency, and sets the gain to 0 dB, which makes the sample value 2^{15} correspond to 1 V output. It is also possible to use $(1, 0)$ as argument to `usrp.selected_subdev` in order to use a daughterboard at side B.

Connect the I and Q transmitter connectors (TX-A and TX-B of the LFTX daughterboard) to channels 1 and 2 of the oscilloscope. Make sure that nothing is connected to the receiver (the LFRX daughterboard) at this time. Study the signal on the oscilloscope and verify that it has amplitude 0.1 V and frequency close to the expected 440 Hz. You may see the tone on both of the channels, even though the imaginary part of the source signal is zero. This is because the CORDIC in the AD9862 chip imposes a complex phase shift of the signal. Unfortunately, it is not possible to clear the phase accumulator register (other than by restarting the USRP), so tuning to 0 Hz will produce a complex output even if the input is real. Hence, if you do this as the first thing you do after powering up the USRP, then the signal should only be available on the I connector.

To see the effect of the complex phase shift, set the carrier frequency (i.e., the variable `carrier`) to 1 Hz and rerun the program. Next, set the carrier frequency to 1 MHz and explain the results.

Note: You can also use a probe to connect channel 4 of the oscilloscope to pin `io_tx(0)` of connector `J50` on the LFTX daughterboard, and use that to trigger the oscilloscope.

4.3 Spectrum Analyzer using USRP

Now connect the I and Q transmitter connectors (TX-A and TX-B of the LFTX daughterboard) to the I and Q receiver connectors (RX-A and RX-B of the LFRX daughterboard) via the two channels of the oscilloscope. The amplitude of the measured signal should then drop to half, since the output and input impedances of the daughterboards are both 50 Ω.

The USRP and GNU radio can be used as a spectrum analyzer. A general analyzer built on the old wxgui GUI library is included in the GNU Radio distribution with the name `usrp_fft.py`. Set the transmitter carrier frequency to 100 kHz and run in from one terminal window. Then you open another terminal and run the command

```
$ /usr/local/gnuradio-3.4.2/bin/usrp_fft.py -d 64 -f 0
```

to see the spectrum of the received signal. Explain the results.

The frequency axis can be changed by instead running

```
$ /usr/local/gnuradio-3.4.2/bin/usrp_fft.py -d 64 -f 100000
```

What would this represent?

4.4 Building a Receiver

In this task, you will build a receiver for the signal generated with `usrp_tx.py`. The USRP A/D conversion rate is fixed at 64 MHz, and you need to bring this down to the 48 kHz that the audio card can sink. The FPGA firmware supports any even integer decimation factor in the range $[4, 256]$. In the preparations to this lab, you should compute a suitable FPGA decimation factor for the USRP, and then compute the additional decimation/interpolation needed at the host.

Call the receiver program `usrp_rx.py`. You will have use for (some of) the following blocks:

```
usrp.source_c
gr.complex_to_real
gr.complex_to_imag
gr.multiply_const_ff
blks2.rational_resampler_fff
audio.sink
```

Use the same `subdev_spec` as you did in `usrp_tx.py`. Change the carrier frequency back to 0 Hz at both the transmitter and the receiver, then run the flowgraph. This should produce the 440 Hz tone in the speakers.

Note: There is no phase synchronization between the transmitter and receiver, and hence the received signal will not necessarily be real. However, we know that the source signal was real. The goal in this task is not to produce a perfect receiver. Therefore, you can allow yourself to just use `complex_to_real` (and skip `complex_to_imag`) and hope for the best.

Note: If the produced audio signal has the wrong frequency, combined with that you notice a "stutter" or "buzzing" of the audible signal, then most likely you need to check your interpolation and decimation rates used by the rational resampler. You would then also notice print-outs like repeated 'Uo' in the terminal. The reason for this is that the flowgraph contains two blocks with fixed sampling rates. The rational resampler has to do the correct thing to ensure that the sampling rates match. Even if you have done everything correctly, you may still see spurious printouts of that kind in the terminal. The reason for that is that the clocks of the USRP and the audio sink are not locked. Eventually that leads to overruns or underruns in the software or hardware buffers, which is what the printouts are reporting. Building a rate-adaptor block to compensate for this is possible, but outside the scope of this introductory lab.

4.5 A USRP equalizer

In the last part of this lab, you will change the sound source from a simple sinus signal to a music sample read from a wave file. Begin by copying `usrp_tx.py` to `wav_tx.py` and `usrp_rx.py` to `wav_rx.py`. Modify `wav_tx.py` to use `gr.wavfile_source` instead of `gr.sig_source_f`:

```
self.src = gr.wavfile_source('test.wav', True)
```

`gr.wavfile_source` produces float samples, so no type conversion is needed.

In `wav_rx.py`, insert three FIR filters as shown in Figure 2, one each for bass, mid, and treble. You may use the following helper functions to design the filters:

```
gr.firdes.low_pass
gr.firdes.band_pass
gr.firdes.high_pass
```

They return tap values as a list, which is then used as the argument `taps` when instantiating the filter with

```
gr.fir_filter_fff(decim_rate,taps)
```

where `decim_rate` should be 1 in this case. Suitable bandwidths and transition bands are given in the following table:

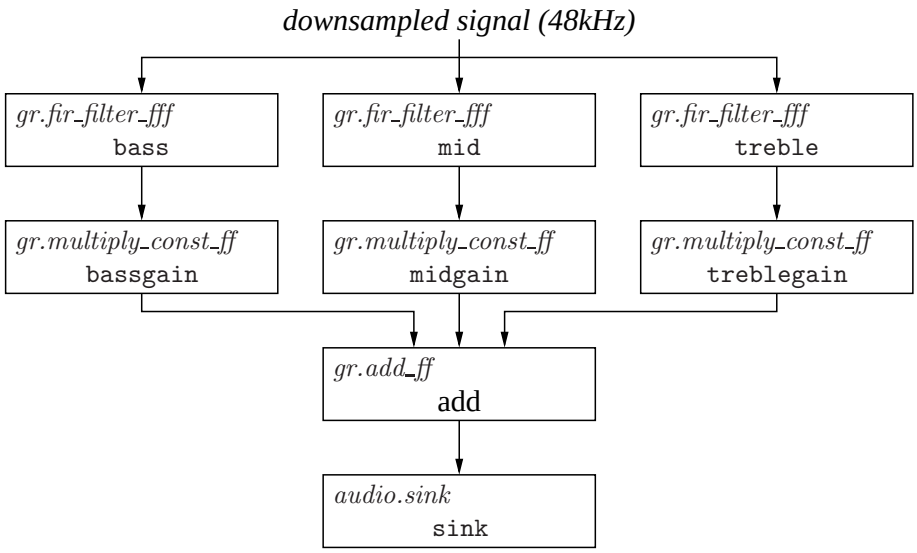

Figure 2: USRP audio player with an equalizer

Filter	Bandwidth	Transition width
Bass	0–100	20
Mid	100–2000	100
Treble	2000–24000	400

Replace the statement starting the flowgraph with the following code to provide a console user interface for setting the equalizer parameters:

```
tb.start()
quit = False
print "Commands: "
print "  bass: b <gain>"
print "  mid: m <gain>"
print "  treble: t <gain>"
print "  quit: q"
while quit == False:
    line = raw_input('> ')
    cmd = line.split()
    if cmd[0] == 'q':
        quit = True
    else:
        arg = float(cmd[1])
        if cmd[0] == 'b':
            tb.bassgain.set_k(arg)
        elif cmd[0] == 'm':
            tb.midgain.set_k(arg)
        elif cmd[0] == 't':
            tb.treblegain.set_k(arg)
tb.stop()
```

5 Resources

- Online USRP FAQ:
  `https://gnuradio.org/redmine/projects/gnuradio/wiki/FAQ`
- Firas Abbas Hamza, *The USRP under 1.5X Magnifying Lens!*:
  `http://gnuradio.org/redmine/attachments/129`
  Third-party documentation of USRP.
- GNU Radio C++ block documentation:
  `file:///usr/local/gnuradio-3.4.2/share/doc/gnuradio-3.4.2/html/index.html`
- Firas Abbas, *Simple User Manual for Gnuradio*:
  `https://www.csun.edu/~skatz/katzpage/sdr_project/sdr/Simple-Gnuradio-User-Manual-v1.0.pdf`
  Third-party documentation of GNU Radio. This one is a bit out of date, but still useful. A printed copy is available at the lab desk.